LORA Hyperparameters

LoRA (Low-Rank Adaptation) is a technique used to fine-tune large language models in a parameter-efficient way. Below is a table outlining common LoRA hyperparameters.

Hyperparameter	Meaning
`r`	Rank of the low-rank matrices used for adaptation. It determines the size of the adaptation matrices and controls the degree of parameter efficiency. Higher `r` allows the model to learn more specific features, while lower `r` leads to more parameter-efficient adaptation.
`alpha`	Scaling factor for the low-rank updates. It controls how much the learned low-rank matrices are scaled before they are added to the original model parameters.
`dropout`	Dropout probability applied during training to prevent overfitting. It is applied to the low-rank matrices or activations in the model.
`bias`	Whether to include a bias term in the adaptation matrices. If set to `True`, the adaptation matrices will have a bias component added to them.
`lora_weights`	The weights in the low-rank adaptation matrices. These weights are fine-tuned while the original model parameters remain frozen.
`layer_norm`	Whether to apply Layer Normalization to the low-rank matrices to stabilize training.

Training Hyperparameters

Training hyperparameters refer to settings that control the training process of a model, affecting the learning rate, optimization, and how the model converges. Below is a table with some common training hyperparameters.

Hyperparameter	Meaning
`learning_rate`	The rate at which the model's parameters are updated during training. A higher rate leads to faster convergence but might risk overshooting the optimal point, while a lower rate allows for more fine-tuned learning.
`batch_size`	The number of training samples processed together in one pass through the model. Larger batches make the training more stable but require more memory.
`epochs`	The number of complete passes through the entire training dataset. More epochs can lead to better model performance but also increase the risk of overfitting.
`optimizer`	The algorithm used to update the model's weights during training. Common optimizers include SGD, Adam, and RMSprop.
`momentum`	A factor used in momentum-based optimization techniques to accelerate convergence by moving the model's parameters in the direction of the past gradients.

`weight_decay`	Regularization term added to the loss function to prevent overfitting by penalizing large weights.
`scheduler`	Learning rate scheduler that adjusts the learning rate during training. It helps to improve convergence and prevent overshooting by reducing the learning rate over time.

---

3. Precision, Recall, Confusion Matrix, F1, ROC-AUC

These terms are commonly used in classification tasks to evaluate the performance of a model.

Precision, Recall, F1, and Confusion Matrix

Metric	Meaning
Precision	The fraction of relevant instances retrieved by the model. It is the ratio of true positives (TP) to the sum of true positives and false positives (FP). Precision=TP/(TP+FP)
Recall	The fraction of relevant instances that were retrieved. It is the ratio of true positives to the sum of true positives and false negatives (FN). Recall={TP}/{TP + FN}
F1 Score	The harmonic mean of precision and recall. It provides a balance between both, especially when the classes are imbalanced. F1=2× 1/ {1/Precision}+{1/Recall}}
Confusion Matrix	A table used to describe the performance of a classification model. It contains the following components: - **True Positive (TP)**: Correctly predicted positive instances - **True Negative (TN)**: Correctly predicted negative instances - **False Positive (FP)**: Negative instances incorrectly predicted as positive - **False Negative (FN)**: Positive instances incorrectly predicted as negative

ROC-AUC

Metric	Meaning
ROC (Receiver Operating Characteristic)	A curve plotting the True Positive Rate (Recall) against the False Positive Rate (FPR) at different thresholds. It shows how well the model distinguishes between classes.
AUC (Area Under the Curve)	The area under the ROC curve. AUC values range from 0 to 1, with a higher AUC indicating better model performance. An AUC of 1 means perfect classification, while an AUC of 0.5 suggests random guessing.

2-a

Concept	Category	Description
BERT Tokenizer	**NLP / Tokenization**	Tokenizes input text into tokens compatible with BERT's vocabulary.
BERT for Sequence Classification	**Pre-trained Model**	Uses BERT for binary sentiment classification.

PEFT (LoRA) Fine-Tuning	**Model Adaptation**	Efficiently fine-tunes large pre-trained models (like BERT) using a low-rank adaptation approach.
SentimentDataset	**Data Preparation**	Custom PyTorch `Dataset` to prepare tokenized inputs for training, validation, and testing.
DataLoader	**Data Handling**	Loads training, validation, and test data in batches.
Train-Test-Validation Split	**Data Preparation**	Splits IMDb dataset into training, validation, and test subsets.
CrossEntropyLoss	**Loss Function**	Calculates the loss between predicted logits and true labels.
AdamW Optimizer	**Optimization Algorithm**	Optimizes model parameters using AdamW (Adam with weight decay).
Device Allocation	**Hardware Acceleration**	Dynamically allocates GPU (if available) or CPU for training.
Training Loop	**Model Training**	Iteratively trains the model over multiple epochs, updating weights using backpropagation.
Evaluation Metrics	**Model Evaluation**	Uses accuracy and classification report (precision, recall, F1-score) for model evaluation.
Saving and Loading Model	**Model Persistence**	Saves and loads trained model weights using PyTorch `.pth` files.
Prediction Function	**Model Inference**	Classifies new input text as "Positive" or "Negative."
Loss and Accuracy Plot	**Visualization**	Plots training and validation loss to visualize model convergence.

Explanation of Each Part

---

1. Environment Setup

- **Purpose:** Sets up the necessary libraries, device, and environment for training.
- **Key Operations:**
  - Import necessary libraries like `torch`, `transformers`, `datasets`, `sklearn`, etc.
  - Check for the availability of a GPU and set the `device` to either "cuda" (GPU) or "cpu".

---

2. Data Loading and Splitting

- **Purpose:** Load the IMDb dataset and split it into training, validation, and test sets.
- **Key Operations:**
  - **Load Dataset:** Uses the `datasets` library to load the IMDb dataset.
  - **Sampling and Splitting:**

        - Samples 10% of the training data to reduce computational load.
        - Splits the test data into 80% test and 20% validation using `random_split`.
    - **Data Organization:** Separates text and labels for easy usage in the `SentimentDataset`.

---

3. Tokenization and Data Preparation

- **Purpose:** Prepares the data for input to the BERT model.
- **Key Operations:**
    - **Custom Dataset Class (`SentimentDataset`)**:
        - Tokenizes text using the BERT tokenizer.
        - Pads and truncates text to a fixed length (128 tokens) to ensure uniform input size.
        - Converts text into PyTorch tensors for model training.
    - **DataLoader Creation**:
        - Batches the data for training, validation, and testing.
        - Uses PyTorch's `DataLoader` to efficiently handle mini-batches.

---

4. Model Definition (BERT with PEFT LoRA)

- **Purpose:** Define the classification model with BERT as the base and LoRA for parameter-efficient fine-tuning.
- **Key Operations:**
    - **BERT Model:** Loads a BERT model pre-trained for sequence classification (`BertForSequenceClassification`).
    - **LoRA Configuration:**
        - Uses the **PEFT (Parameter-Efficient Fine-Tuning)** technique, which adds trainable low-rank layers to BERT, making fine-tuning more efficient.
        - The LoRA configuration specifies the rank (`r=8`), alpha, and dropout rate for LoRA layers.

---

5. Loss Function and Optimizer

- **Purpose:** Define the loss function and optimizer for training.
- **Key Operations:**
    - **Loss Function**: Uses `CrossEntropyLoss` to compute the difference between the predicted logits and true labels.
    - **Optimizer**: Uses `AdamW`, a variant of Adam that incorporates weight decay to reduce overfitting.

---

6. Training Loop

- **Purpose:** Train the BERT model using backpropagation.
- **Key Operations:**
    - **Loop Structure**:
        - For each epoch, the model trains on the batches of training data.
        - It computes the loss, backpropagates the error, and updates model weights using the optimizer.
    - **Validation Loop**:
        - After training on each epoch, the model is evaluated on the validation set.
        - It computes validation loss and accuracy, printing them for every epoch.
    - **Loss Tracking**:

        - Records training and validation loss for visualization.

---

7. Model Saving and Loading

- **Purpose:** Save and load the trained model for reuse or deployment.
- **Key Operations:**
    - **Save Model**: Saves the model's state dictionary (weights) as `bert_sentiment_model.pth`.
    - **Load Model**: Reloads the model with the saved weights and LoRA configuration.

---

8. Evaluation on the Test Set

- **Purpose:** Evaluate the model on the test set to assess final model performance.
- **Key Operations:**
    - **Prediction**:
        - Loops through test batches, makes predictions using the trained model, and computes accuracy.
    - **Classification Report**:
        - Uses `classification_report` from `sklearn` to compute precision, recall, and F1-score for both positive and negative classes.

---

9. Inference on Sample Texts

- **Purpose:** Demonstrate the model's ability to predict sentiments for unseen text.
- **Key Operations:**
    - **Tokenization**: Tokenizes the input text using BERT's tokenizer.
    - **Prediction**: Uses the trained model to classify the input as **Positive** or **Negative**.
    - **Examples**: Three example texts are classified to demonstrate the model's sentiment classification capability.

---

10. Loss Plot (Training vs. Validation)

- **Purpose:** Visualize the model's training and validation loss over epochs.
- **Key Operations:**
    - **Plot Loss**: Plots the loss curves for both training and validation, allowing users to observe the model's convergence and possible overfitting.

Summary of Flow

1. **Data Handling**: Loads, tokenizes, and batches the IMDb dataset into training, validation, and test splits.
2. **Model Design**: Defines a BERT model with LoRA fine-tuning for sentiment classification.
3. **Training**: Trains the model over 3 epochs using mini-batch training.
4. **Evaluation**: Evaluates the model on the validation and test sets using accuracy, precision, recall, and F1-score.
5. **Inference**: Uses the trained model to predict sentiment on unseen reviews.
6. **Visualization**: Plots the training and validation loss to visualize model convergence.

Table of Key Concepts

Concept	Description
Device Setup	Check for GPU availability and set the computation device.
Data Handling	Load and split IMDB dataset into train, validation, and test sets.
Data Preparation	Create a custom dataset class for tokenizing input texts.
Model Selection	Use GPT-2 with LoRA (Low-Rank Adaptation) for sentiment classification.
PEFT LoRA	Efficiently fine-tune GPT-2 with LoRA configuration.
Loss & Optimizer	Use CrossEntropyLoss and AdamW optimizer for training.
Training Loop	Train the model while tracking training and validation loss.
Evaluation	Evaluate the model on the test set and display accuracy and classification report.
Model Save/Load	Save and reload the trained model from a local file.

Step-by-Step Explanation

1. **Device Setup**
   - The device is set to GPU (`cuda`) if available; otherwise, it defaults to CPU. This allows the model to leverage hardware acceleration when available.
2. **Data Handling**
   - The IMDB dataset is loaded using the `datasets` library.
   - A 10% sample of the training set is selected randomly to reduce training time.
   - The test set is split into an 80:20 ratio to create a validation set.
   - Training, validation, and test data are separated into text (`X_train`, `X_val`, `X_test`) and labels (`y_train`, `y_val`, `y_test`).
3. **Data Preparation**
   - A **custom** **`SentimentDataset`** class is created to tokenize input texts using GPT-2's tokenizer.
   - The class processes each text into `input_ids` and `attention_mask` to ensure compatibility with the model.
   - **DataLoaders** are created to load batches of data for training, validation, and testing.
4. **Model Selection**
   - **GPT-2 for Sequence Classification** is used as the base model, and LoRA (Low-Rank Adaptation) is applied to reduce the number of trainable parameters.
   - The model's configuration is updated to use `eos_token` as the padding token.
5. **PEFT (LoRA) Configuration**
   - LoRA is applied to **inject learnable parameters into specific layers** of GPT-2.
   - **LoRA Config** parameters:
     - `r=8`: Rank of the low-rank matrices.
     - `lora_alpha=32`: Scaling factor.
     - `lora_dropout=0.1`: Dropout to avoid overfitting.
     - `task_type="SEQ_CLS"`: Specifies the task as sequence classification.
6. **Loss Function & Optimizer**
   - **CrossEntropyLoss** is used as the loss function since this is a binary classification problem.

    - **AdamW optimizer** is used to update the model's weights with a learning rate of `2e-5`.
7. **Training Loop**
    - The model is trained for **3 epochs**.
    - Each epoch tracks training loss and validation loss, which is printed after every epoch.
    - During training, **backpropagation** is used to update the model's parameters.
    - During validation, the model's accuracy and loss are recorded, and predictions are made using the validation set.
8. **Evaluation on Test Set**
    - The model is evaluated on the test set.
    - Predictions are compared with true labels to compute **test accuracy**.
    - A **classification report** is generated, providing precision, recall, and F1-score for each class.
9. **Plot Training and Validation Loss**
    - **Training and validation loss** is plotted to visualize model convergence.
    - This helps assess potential overfitting or underfitting issues.
10. **Save & Reload the Model**
    - The model's weights are saved to a `.pth` file.
    - The model can be reloaded and used for inference without retraining.